National Textile University

Department of Computer Science

Subject:

Operating System

Submitted to:

Sir Nasir

Submitted by:

Eman Babar

Reg. number:

23-NTU-CS-FL-1148

Semester:

5^{th}- A

LAB-09

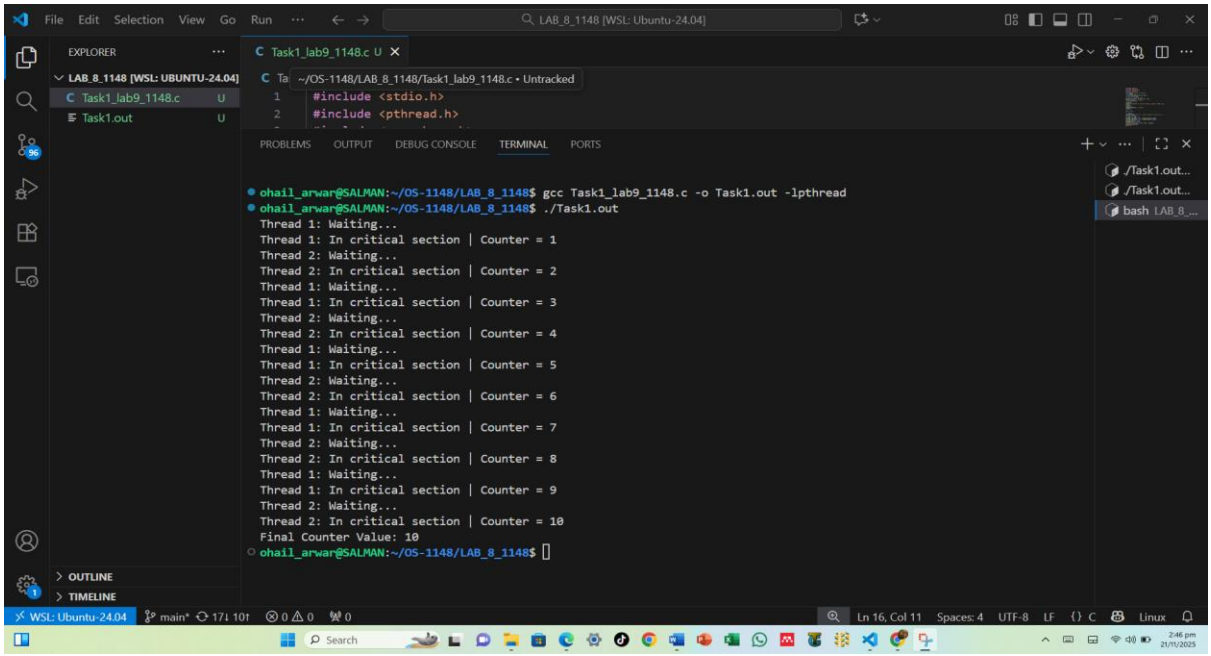
TASK-1-Binary Semaphore

a)

Code:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
int counter = 0;
void* thread_function(void* arg) {
int id = *(int*)arg;
for (int i = 0; i < 5; i++) {
printf("Thread %d: Waiting...\n", id);
sem_wait(&mutex); // Acquire
// Critical section
counter++;
printf("Thread %d: In critical section | Counter = %d\n", id,
counter);
sleep(1);
sem_post(&mutex); // Release
sleep(1);
}
return NULL;
}
int main() {
sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
pthread_t t1, t2;
int id1 = 1, id2 = 2;
pthread_create(&t1, NULL, thread_function, &id1);
pthread_create(&t2, NULL, thread_function, &id2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Final Counter Value: %d\n", counter);
sem_destroy(&mutex);
return 0;
}
```

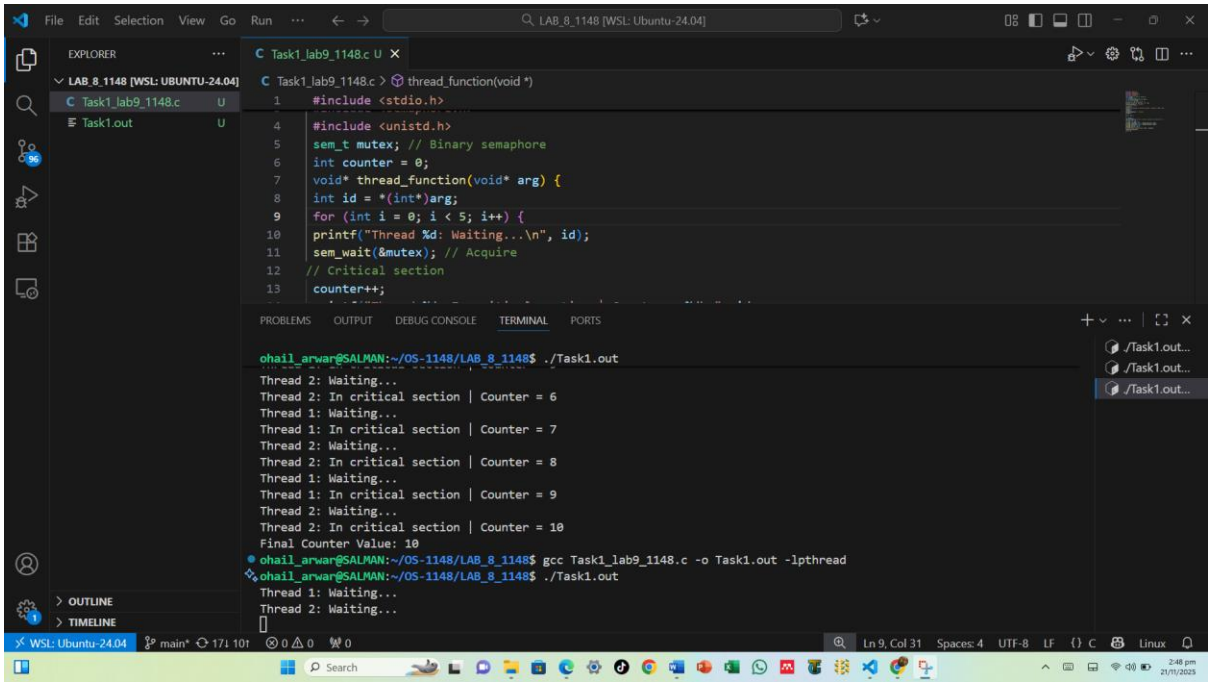
Output:



b)

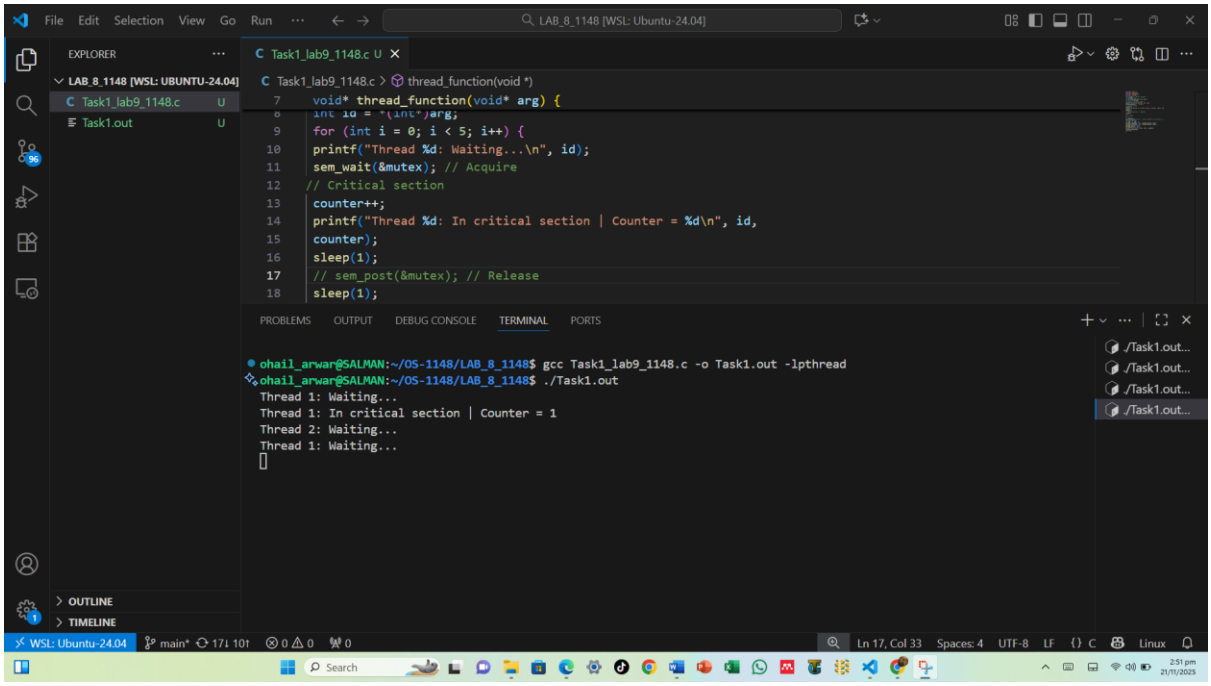
Code:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
int counter = 0;
void* thread_function(void* arg) {
int id = *(int*)arg;
for (int i = 0; i < 5; i++) {
printf("Thread %d: Waiting...\n", id);
sem_wait(&mutex); // Acquire
// Critical section
counter++;
printf("Thread %d: In critical section | Counter = %d\n", id,
counter);
sleep(1);
sem_post(&mutex); // Release
sleep(1);
}
return NULL;
}
int main() {
sem_init(&mutex, 0, 0); // Binary semaphore initialized to 1
pthread_t t1, t2;
int id1 = 1, id2 = 2;
pthread_create(&t1, NULL, thread_function, &id1);
pthread_create(&t2, NULL, thread_function, &id2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Final Counter Value: %d\n", counter);
sem_destroy(&mutex);
return 0;
}
```



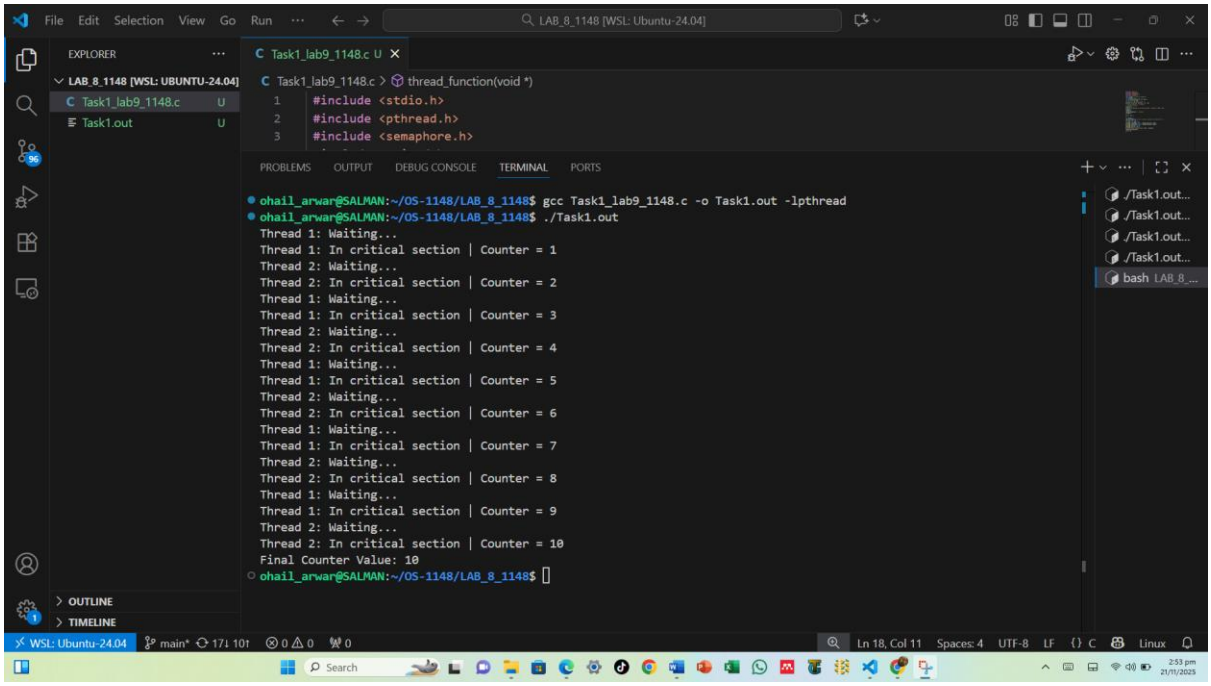
c)

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
int counter = 0;
void* thread_function(void* arg) {
int id = *(int*)arg;
for (int i = 0; i < 5; i++) {
printf("Thread %d: Waiting...\n", id);
sem_wait(&mutex); // Acquire
// Critical section
counter++;
printf("Thread %d: In critical section | Counter = %d\n", id,
counter);
sleep(1);
// sem_post(&mutex); // Release
sleep(1);
}
return NULL;
}
int main() {
sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
pthread_t t1, t2;
int id1 = 1, id2 = 2;
pthread_create(&t1, NULL, thread_function, &id1);
pthread_create(&t2, NULL, thread_function, &id2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Final Counter Value: %d\n", counter);
sem_destroy(&mutex);
return 0;
}
```



d)

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
int counter = 0;
void* thread_function(void* arg) {
int id = *(int*)arg;
for (int i = 0; i < 5; i++) {
printf("Thread %d: Waiting...\n", id);
//sem_wait(&mutex); // Acquire
// Critical section
counter++;
printf("Thread %d: In critical section | Counter = %d\n", id,
counter);
sleep(1);
sem_post(&mutex); // Release
sleep(1);
}
return NULL;
}
int main() {
sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
pthread_t t1, t2;
int id1 = 1, id2 = 2;
pthread_create(&t1, NULL, thread_function, &id1);
pthread_create(&t2, NULL, thread_function, &id2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Final Counter Value: %d\n", counter);
sem_destroy(&mutex);
return 0;
}
```



Comments:

In this code, semaphore ensures that only one thread enters the critical section at a time while incrementing the shared counter. Then, two threads wait and then update counter safely, print their status and release semaphore. After both threads finish, the final counter value is printed. When we initialize semaphore to 0 then it will block both threads forever so they never enter critical section and as a result, program never ends. And, if we comment "sem_post(&mutex);)), the semaphore is never released, so only the first thread enters once and then both threads block forever. And, if we comment "sem_wait(&mutex);)), both threads enter the critical section at the same time which may cause race conditions.

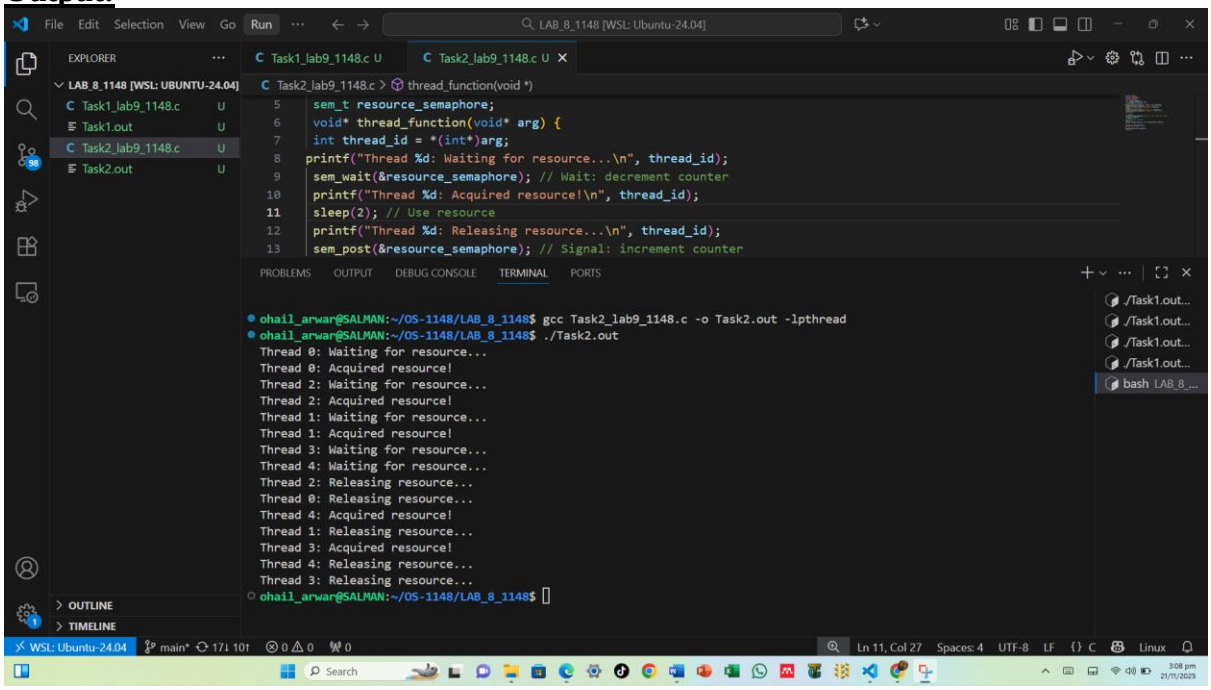
TASK-2-Counting Semaphore

a)

Code:

```c
#include <stdio.h>
 #include <pthread.h>
 #include <semaphore.h>
 #include <unistd.h>
 sem_t resource_semaphore;
 void* thread_function(void* arg) {
 int thread_id = *(int*)arg;
printf("Thread %d: Waiting for resource...\n", thread_id);
 sem_wait(&resource_semaphore); // Wait: decrement counter
 printf("Thread %d: Acquired resource!\n", thread_id);
 sleep(2); // Use resource
 printf("Thread %d: Releasing resource...\n", thread_id);
 sem_post(&resource_semaphore); // Signal: increment counter
 return NULL;
 }
 int main() {
 sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
 pthread_t threads[5];
 int ids[5];
 for (int i = 0; i < 5; i++) {
 ids[i] = i;
 pthread_create(&threads[i], NULL, thread_function, &ids[i]);
 }
 for (int i = 0; i < 5; i++) {
 pthread_join(threads[i], NULL);
 }
 sem_destroy(&resource_semaphore);
 return 0;
 }
```

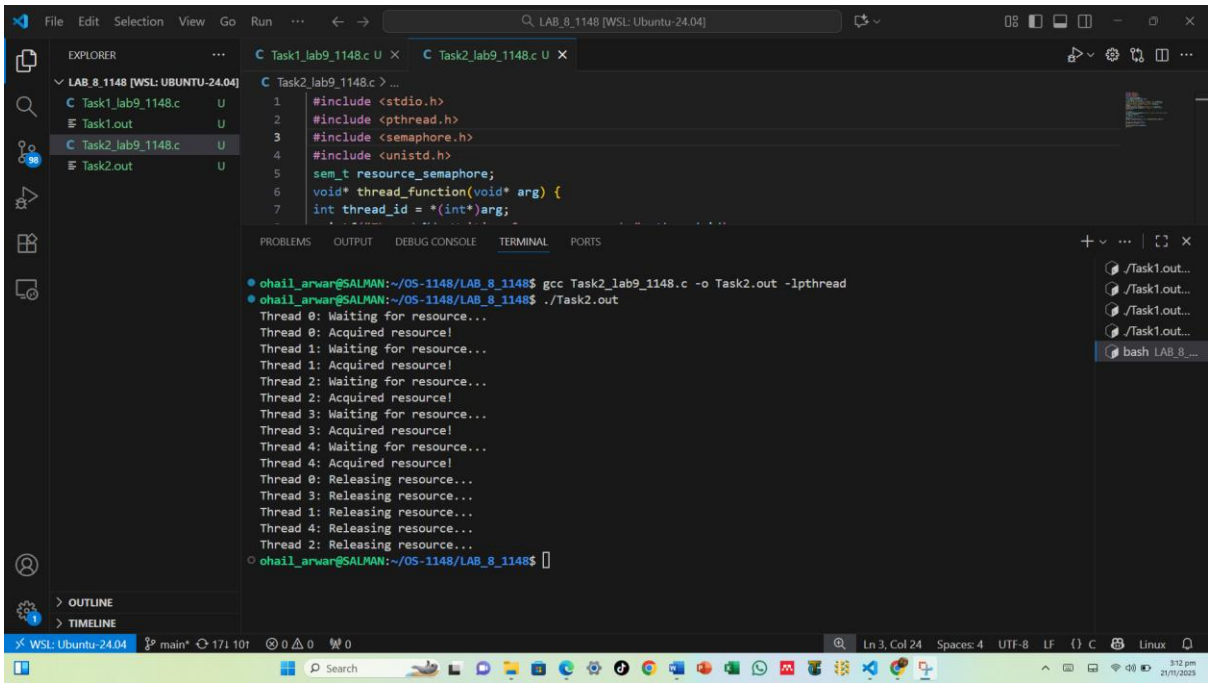
Output:



b)

Code:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t resource_semaphore;
void* thread_function(void* arg) {
int thread_id = *(int*)arg;
printf("Thread %d: Waiting for resource...\n", thread_id);
//sem_wait(&resource_semaphore); // Wait: decrement counter
printf("Thread %d: Acquired resource!\n", thread_id);
sleep(2); // Use resource
printf("Thread %d: Releasing resource...\n", thread_id);
sem_post(&resource_semaphore); // Signal: increment counter
return NULL;
}
int main() {
sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
pthread_t threads[5];
int ids[5];
for (int i = 0; i < 5; i++) {
ids[i] = i;
pthread_create(&threads[i], NULL, thread_function, &ids[i]);
}
for (int i = 0; i < 5; i++) {
pthread_join(threads[i], NULL);
}
sem_destroy(&resource_semaphore);
return 0;
}
```

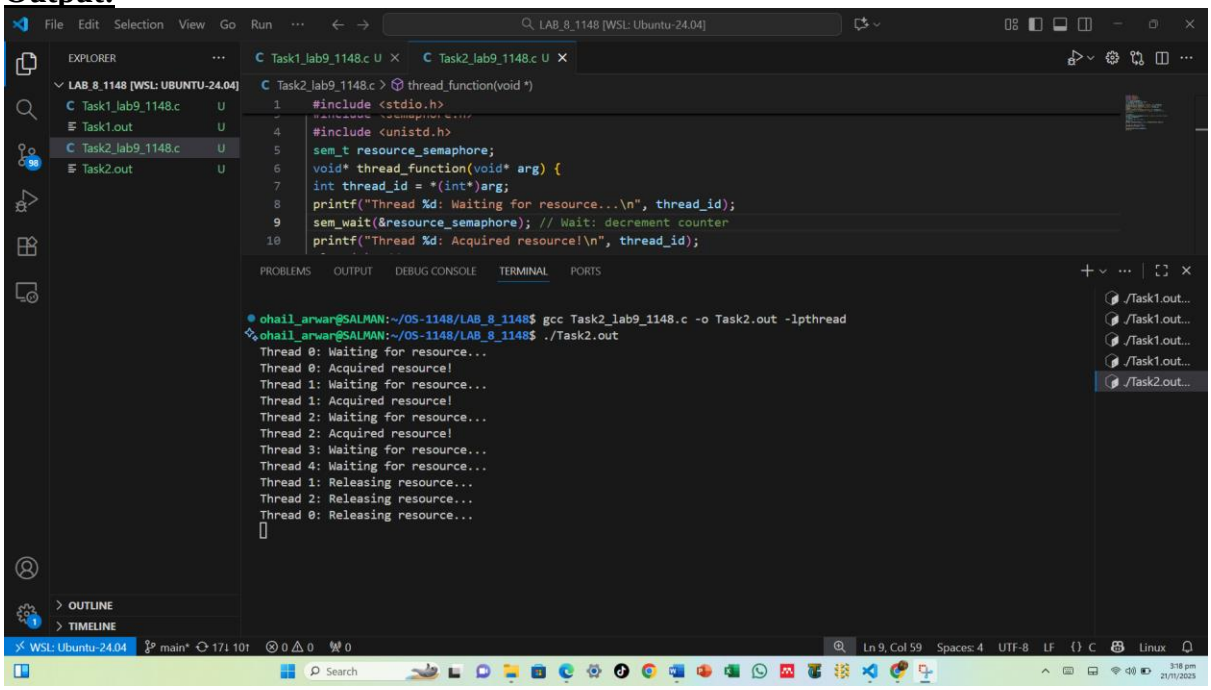


c)

Code:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t resource_semaphore;
void* thread_function(void* arg) {
int thread_id = *(int*)arg;
printf("Thread %d: Waiting for resource...\n", thread_id);
sem_wait(&resource_semaphore); // Wait: decrement counter
printf("Thread %d: Acquired resource!\n", thread_id);
sleep(2); // Use resource
printf("Thread %d: Releasing resource...\n", thread_id);
//sem_post(&resource_semaphore); // Signal: increment counter
return NULL;
}
int main() {
sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
pthread_t threads[5];
int ids[5];
for (int i = 0; i < 5; i++) {
ids[i] = i;
pthread_create(&threads[i], NULL, thread_function, &ids[i]);
}
for (int i = 0; i < 5; i++) {
pthread_join(threads[i], NULL);
}
sem_destroy(&resource_semaphore);
return 0;
}
```

Output:



d)

Code:

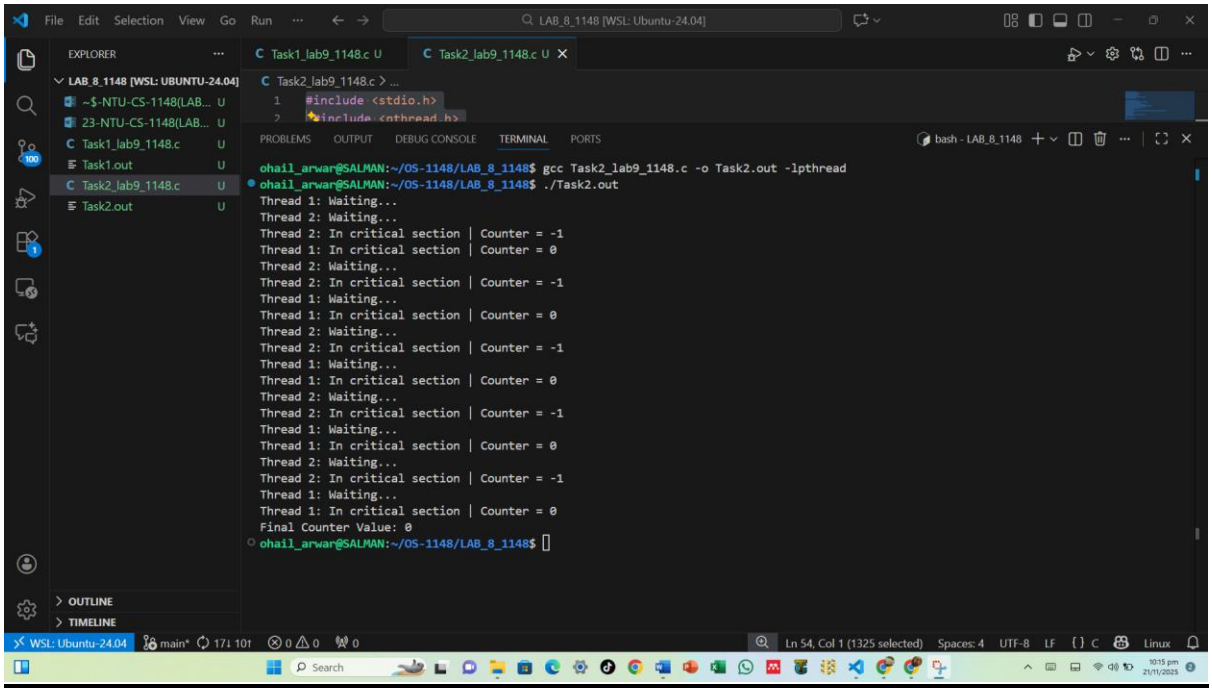
```c
#include <stdio.h>
 #include <pthread.h>
 #include <semaphore.h>
 #include <unistd.h>
 sem_t mutex; // Binary semaphore
 int counter = 0;

 //FUNCTION FOR THREAD
 void* thread_function1(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 5; i++) {
    printf("Thread %d: Waiting...\n", id);
    sem_wait(&mutex); // Acquire
    // Critical section
    counter++;
    printf("Thread %d: In critical section | Counter = %d\n", id,
    counter);
    sleep(1);
    sem_post(&mutex); // Release
    sleep(1);
 }
    return NULL;
 }

 // FUNCTION FOR THREAD2

  void* thread_function2(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 5; i++) {
    printf("Thread %d: Waiting...\n", id);
    sem_wait(&mutex); // Acquire
    // Critical section
    counter--;
    printf("Thread %d: In critical section | Counter = %d\n", id,
    counter);
    sleep(1);
    sem_post(&mutex); // Release
    sleep(1);
 }
    return NULL;
 }
 int main() {
    sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;
    pthread_create(&t1, NULL, thread_function1, &id1);
    pthread_create(&t2, NULL, thread_function2, &id2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final Counter Value: %d\n", counter);
    sem_destroy(&mutex);
    return 0;
 }
```

Output:



Comments:

This program uses a counting semaphore which is initialized to 3, and allow only 3 threads to access the shared resource at the same time, while the remaining 2 threads wait until a slot becomes free. If we comment sem_wait(&resource_semaphore), all five threads enter the resource section together with no limit, so the semaphore provides no control. If we comment sem_post(&resource_semaphore), threads will never release the resource and cause only first three threads to run while remaining two stays blocked forever. With one added function, both threads safely increments or decrements the shared counter using a binary semaphore so only one thread enters the critical section at a time.

TASK-3-Comparison of Mutex lock and Binary Semaphore

Similarities:

- Both are used for protecting shared resources in multithreading.
- Both allow only one thread to access a critical section at a time.
- Both help in preventing race conditions and data consistency.

Differences:

Features	Mutex Lock	Binary Semaphore
Ownership	Only the thread that locks it can unlock it.	there is no ownership, any thread can signal (post) it.
Purpose	Designed only for mutual exclusion i.e., locking.	Designed for both mutual exclusion and signalling between threads.
Blocking	If already locked, thread waits until released.	If value is 0 then waits, but if value is 1 then it enters immediately.
Speed	Slightly faster	Slightly slower
Usage Restriction	Always used in pairs i.e., lock and unlock.	It is used with wait (sem_wait) and post (sem_post).